实 验 报 告

**课程名称：** 计算机图形学

**实验仪器：** 计算机

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **项目** | **报告格式**  **Report format** | **代码质量及完成度**  **Code quality** | **逻辑或思想描述**  **Necessitate logical description** | **独创性**  **Originality** | **合计**  **Total** |
| **百分比(%)**  **percentage** | **15** | **30** | **35** | **20** | **100** |
| 得分（score） |  |  |  |  |  |

**系 别： 计算机系**

**专 业： 计算机科学与技术**

**班级姓名： 计科1905\陈浩**

**学 号： 2019010132**

**日 期： 2021/9/20 ~ 2021/11/13**

**成 绩：**

**指导教师：** 丁 濛

**实验项目：**直线扫描转换算法

# 实验目的(Objects)

1. 实现直线扫描转换中的DDA和中点画线算法；
2. 理解中点画线算法的实质。

# 实验内容 (Contents)

1. 配置必要的编程环境。可以使用任何一种你熟悉的GUI环境，包括但不限于 Java，MFC、QT、Objective-C、OpenGL、OpenCV等。
2. 实现完整版的DDA算法（鼠标任意点击两点，然后在窗口中画出两点间的线段），不能使用任何自带的画线函数，只能使用画点函数或是直接对图像的某个像素进行赋值操作；
3. 实现完整版的中点画线算法（鼠标任意点击两点，然后在窗口中画出两点间的线段），不能使用任何自带的画线函数，只能使用画点函数或是直接对图像的某个像素进行赋值操作；
4. （附加题）实现中点画圆算法。

# 实验内容 (Your steps or codes, Results)

1.DDA核心算法（Java）:

public void drawline(int x1,int y1,int x2,int y2){

float x = x1;

float y = y1;

float k = (float) (y2-y1) / (x2-x1);

// k = ∞

if(x1 == x2) {

if(y1 > y2) {

for (int i = y1; i >= y2 ; i--) {

g.fillOval(x2, i, Inch, Inch);

}

}

else {

for (int i = y1; i <= y2; i++) {

g.fillOval(x1, i, Inch, Inch);

}

}

}

// k ∈ [-1, 1]

// y = kx + b

if(Math.abs(k) <= 1){

if(x1 > x2) {

for(int i = x1; i >= x2; i--) {

g.fillOval(i, Math.round(y), Inch, Inch);

y -= k;

}

}

else {

for(int i = x1; i <= x2; i++) {

g.fillOval(i, Math.round(y) ,Inch, Inch);

y += k;

}

}

}

// k ∈ [-∞, -1) ∪ (1, +∞]

// x = 1/k - b/k

else {

if(y1>y2) {

for(int i = y1; i >= y2; i--) {

g.fillOval(Math.round(x), i, Inch, Inch);

x -= 1 / k;

}

}

else {

for(int i = y1; i <= y2; i++){

g.fillOval(Math.round(x),i, Inch, Inch);

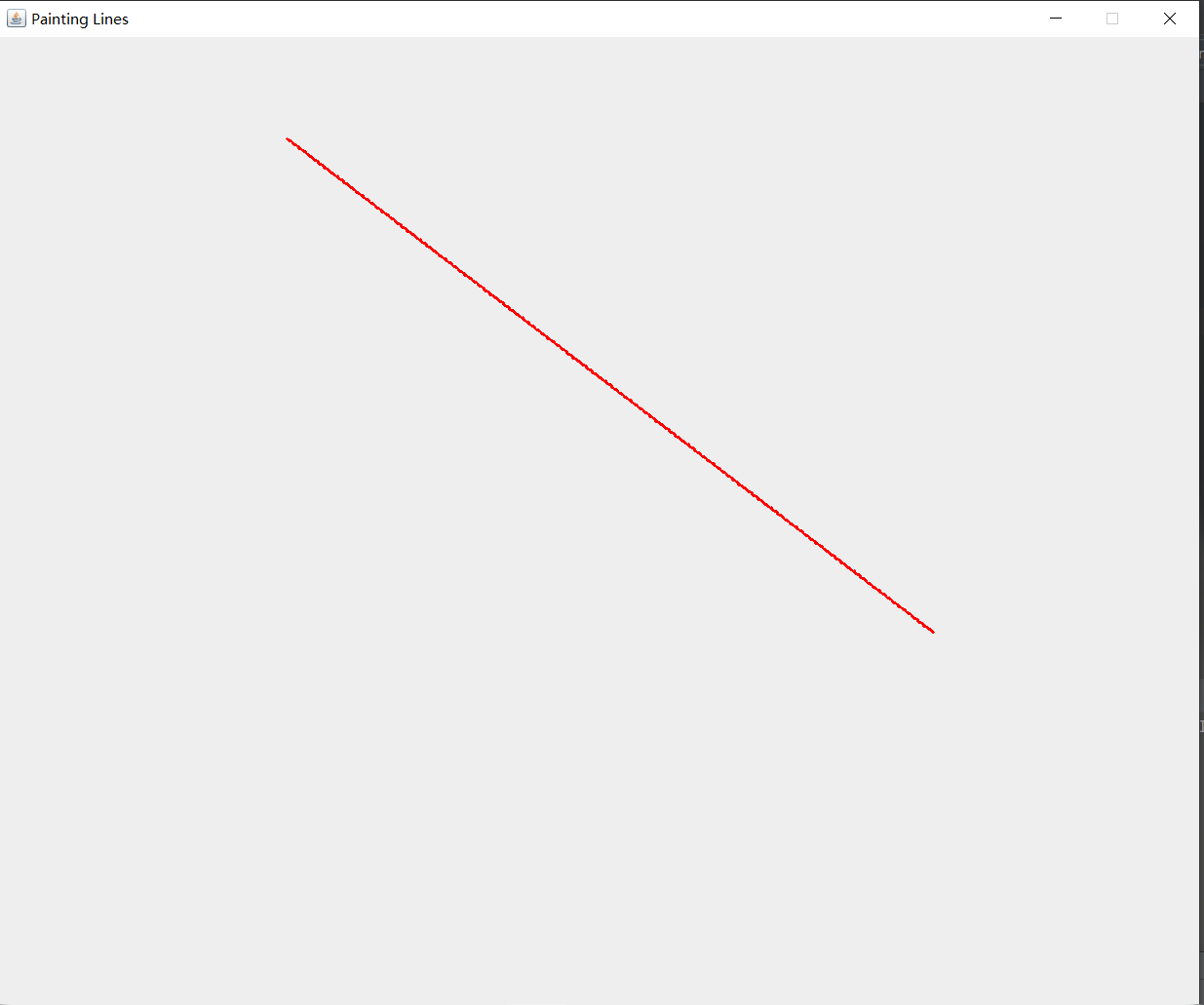
x += 1 / k;

}

}

}

}

结果：

2.中点画圆核心算法（Java）：

public void drawCircle(int x1, int y1, int x2, int y2) {

int x, y, r;

double d;

x = 0; // 特殊x坐标

r = (int) Math.sqrt((Math.pow((y2 - y1), 2)) + (Math.pow((x2 - x1), 2))); // 鼠标点击两点的距离(圆的半径)

y = r; // 特殊y坐标

// 假设P(x, y)已是该段圆弧上实际最接近圆弧的点 则有P1(x + 1, y) (在P的正右侧) 和 P2(x + 1, y - 1) (在P的右下方)

// 若M是P1和P2的中点 即M坐标为(x + 1, y - 0.5)

// 构造函数 F(x) = x² + y² - R²

// 可知 d(M) = (x+1)² + (y-0.5)² - R²

// 有一特殊点[0(±R) , ±R(0)], 可得d0 = 1² + (R - 0.5)² - R² = 1.25 - R 可作为初始圆上距离点!

// d = 1.25 - r;

// Bresenham算法改进: 将d总体 ×2 (消除浮点数)

// 即 d = 2.5 - 2r = 3 - 2r

d = 3 - 2 \* r;

// 确定圆心点:

g.fillOval(x1, y1, CirclePointInch, CirclePointInch);

// 1/8画圆法: 当 x = y 时, 正好完成整个圆的1/8部分

while(x <= y) {

if(d < 0) {

//1.若 d < 0 : 说明M点在圆内 即P1更接近圆弧

// 故下一个判别式为: d' = (x+2)² + (y-0.5)² - R²

// 展开递推可得: d' = d + 2x + 3

// d += 2 \* x + 3;

// Bresenham算法改进: 整体 × 2 (消除浮点数)

// 可得 d' = d + 4x + 6

d += 4 \* x + 6;

}

else{

//2.若 d > 0 : 说明M点在圆内 即P2更接近圆弧

// 故下一个判别式为: d' = (x+2)² + (y-1.5)² - R²

// 展开递推可得: d' = d + 2(x-y) + 5

// d += 2 \* (x - y) + 5;

// Bresenham算法改进: 整体 × 2 (消除浮点数)

// 可得 d' = d + 4(x-y) + 10

d += 4 \* (x - y) + 10;

// 判定结束就将y-1

y--;

}

// 两者判定结束都一定将x+1

x++;

// 分别画出八个45°象限的 1/8 圆:

g.fillOval(x1 + y, y1 - x, EdgeInch, EdgeInch); // 0 ~ 45°

g.fillOval(x1 + x, y1 - y, EdgeInch, EdgeInch); // 45° ~ 90°

g.fillOval(x1 - x, y1 - y, EdgeInch, EdgeInch); // 90° ~ 135°

g.fillOval(x1 - y, y1 - x, EdgeInch, EdgeInch); // 135° ~ 180°

g.fillOval(x1 - y, y1 + x, EdgeInch, EdgeInch); // 180° ~ 225°

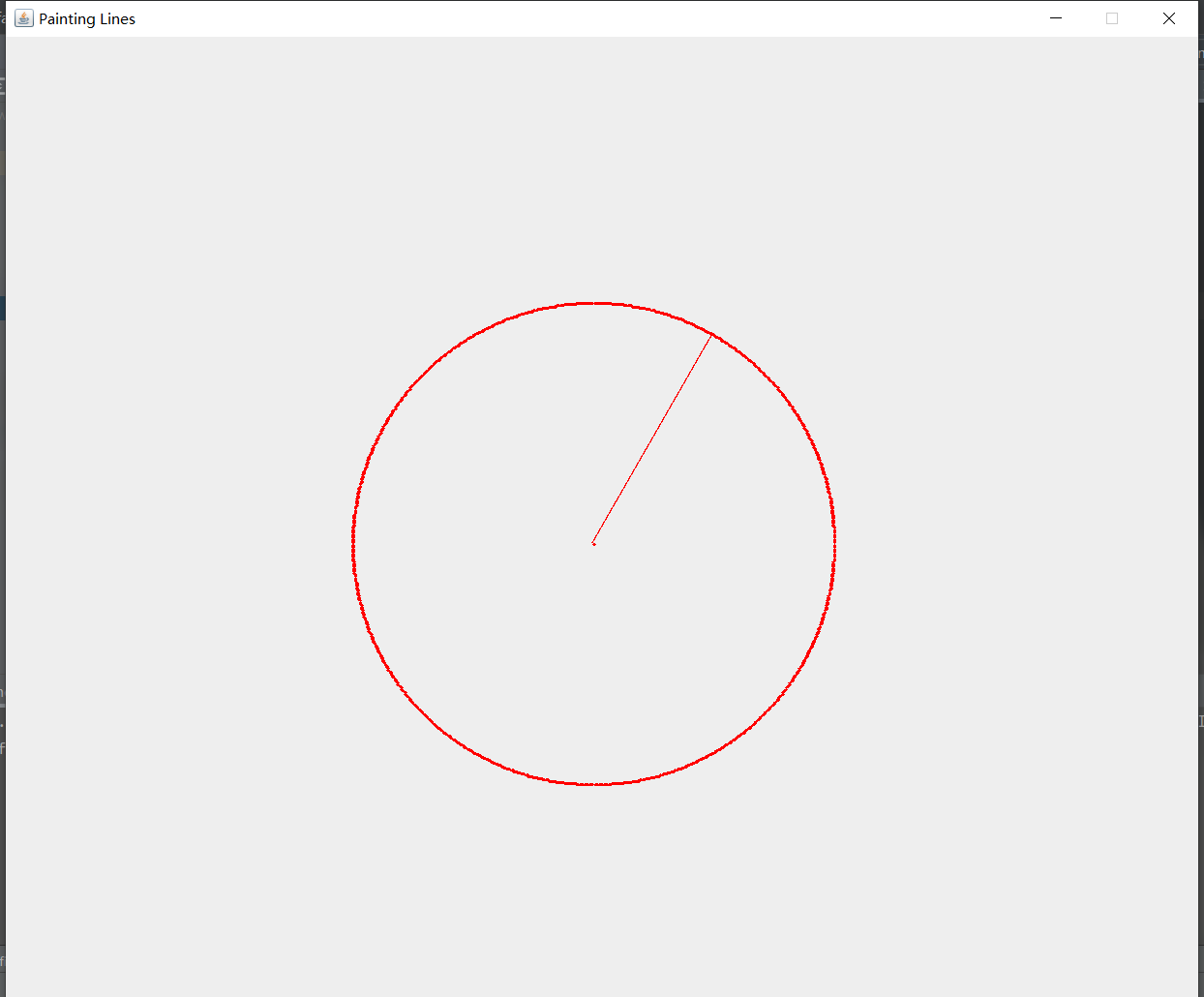
g.fillOval(x1 - x, y1 + y, EdgeInch, EdgeInch); // 225° ~ 270°

g.fillOval(x1 + x, y1 + y, EdgeInch, EdgeInch); // 270° ~ 315°

g.fillOval(x1 + y, y1 + x, EdgeInch, EdgeInch); // 315° ~ 360°

}

}

结果：

# 实验总结 (Conclusion)

本次实验要求完成直线扫描转移算法，由于我对C++掌握并不透彻，故采用Java语言进行编写完成实验。

在直线扫描转移算法中主要分为DDA算法和中点画圆算法，在两个算法中，我均采用了Java.Swing接口进行对画笔和画板的创建，并通过重写主要实现方法drawLine()和drawCircle()完成对鼠标操作进行描点画图。

其中DDA算法中采用了以常量b不变动，改变变量k的方法，分别对k∈[-1，1]和k∈(-∞, -1)∪(1, +∞)进行讨论，分别改变y和x的增量进行描点画线，最终可以得出由鼠标事件输入的起始和终止点间的线段。

而中点画圆算法采用了对假设的中点位于圆上、圆内还是圆外进行讨论，从而得出更靠近该段圆弧的坐标点，并采用递归方式持续找点，最终可以连接成为整个圆1/8的圆弧，最后对四个象限分别取点描弧即可得出整个圆的形状。

通过本次实验，我掌握了通过鼠标事件的输入和图形算法的基础，了解了一个图形的微观是由无数个单位点通过特定的算法进行线性相连构成，也加深了我对图形学的认知。

**实验项目：** 区域填充算法

# 实验目的(Objects)

1. 实现种子填充算法。

# 实验内容 (Contents)

1. 实现种子填充算法，通过鼠标，交互的画出一个多边形，然后利用种子填充算法，填充指定的区域。不能使用任何自带的填充区域函数，只能使用画点、画线函数或是直接对图像的某个像素进行赋值操作；
2. （附加题）实现多边形的扫描线填充算法。

# 实验内容 (Your steps or codes, Results)

种子填充核心算法：

// 画线构成多边形

if(time2 != 0) {

for(int Sum = 0;Sum <= index; Sum++) {

if(Sum == index - 1) {

// 最后一个点自动连接

drawLine(a[Sum], b[Sum], a[0], b[0], g2d);

break;

}

else {

// 画线

drawLine(a[Sum], b[Sum], a[Sum+1], b[Sum+1], g2d);

}

}

}

// 获取边界条件

if(time != 0) {

boolean spanNeedFill;

location l = new location();

l.x = a[0] + 10;

l.y = b[0] + 10;

Stack<location> stack = new Stack<>();

// 将第一个点压入栈

stack.push(l);

while(!stack.empty()) {

location lo=stack.pop();

int x = lo.x;

int y = lo.y;

// if(a > 50)

// { System.out.println("开始画上时x"+x);

// System.out.println("开始画上时y"+y);

// }

// System.out.print("y"+y);

while(!allPoint.contains(new Point(x,y)) && !drawPoint.contains(new Point(x,y)))

{

//System.out.print(x);

drawPoint.add(new Point(x,y));

//System.out.println("需画点"+x+","+y);

x++;

}

int xr = x-1;

x = lo.x-1;

while(!allPoint.contains(new Point(x,y)) && !drawPoint.contains(new Point(x,y)))

{

drawPoint.add(new Point(x,y));

//System.out.println("需画点"+x+","+y);

//g2d.drawString(".", x,y);

x--;

}

int xl = x+1;

x = xl;

y = y+1;

//处理上面的扫描线

while(x <= xr) {

spanNeedFill = false;

while(!allPoint.contains(new Point(x,y)) && !drawPoint.contains(new Point(x,y)))

{

spanNeedFill=true;

x++;

}

if(spanNeedFill)

{

location lc = new location();

lc.x = x-1;

lc.y = y;

stack.push(lc);

}

while(allPoint.contains(new Point(x,y)) || drawPoint.contains(new Point(x,y)) && x <= xr)

x++;

}

//下扫描线

x = xl;

y = y-2;

while(x <= xr)

{

spanNeedFill = false;

while(!allPoint.contains(new Point(x,y)) && !drawPoint.contains(new Point(x,y)))

{

spanNeedFill = true;

x++;

}

if(spanNeedFill)

{

lo.x = x-1;

lo.y = y;

stack.push(lo);

}

while(allPoint.contains(new Point(x,y)) || drawPoint.contains(new Point(x,y)) && x <= xr)

x++;

}

}

for (int i = 0; i < drawPoint.size(); i++) {

Point p = drawPoint.get(i);

g2d.drawString(".", p.x, p.y);

}

}

}

public void drawLine(int x0,int y0,int x1,int y1,Graphics2D g) {

int x = x0;

int y = y0;

int w = x1 - x0;

int h = y1 - y0;

int dx1 = w < 0 ? -1: (w > 0 ? 1 : 0);

int dy1 = h < 0 ? -1: (h > 0 ? 1 : 0);

int dx2 = w < 0 ? -1: (w > 0 ? 1 : 0);

int dy2 = 0;

int fastStep = Math.abs(w);

int slowStep = Math.abs(h);

if (fastStep <= slowStep) {

fastStep = Math.abs(h);

slowStep = Math.abs(w);

dx2= 0;

dy2= h < 0 ? -1 : (h > 0 ? 1 : 0);

}

int numerator = fastStep >> 1;

for (int i = 0; i <=fastStep; i++) {

g.drawString(".", x, y);

Point p=new Point(x,y);

allPoint.add(p);

numerator+= slowStep;

if (numerator >= fastStep) {

numerator -= fastStep;

x += dx1;

y += dy1;

}else {

x += dx2;

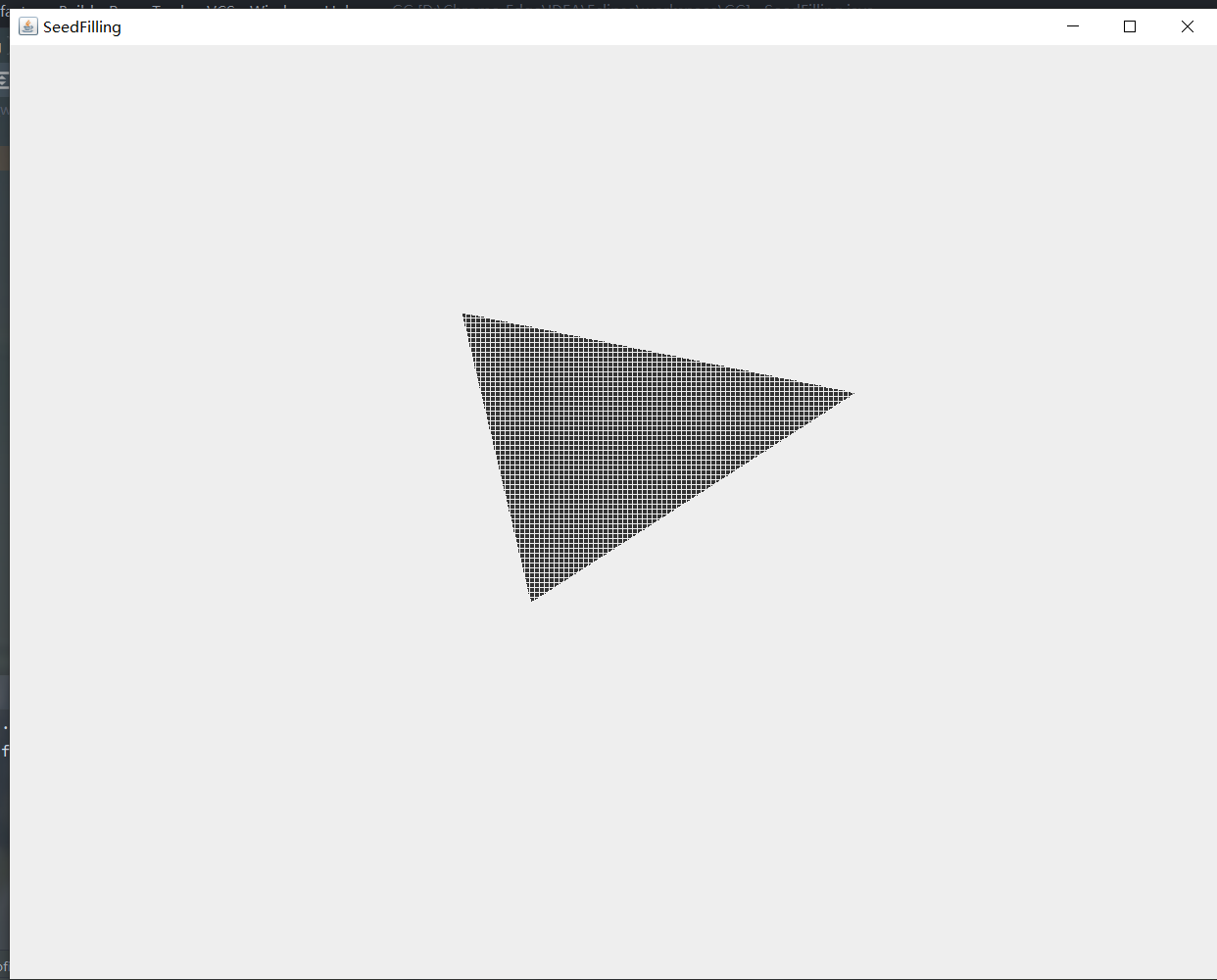
y += dy2;

}

}

}

实验截图：



# 实验总结 (Conclusion)

种子填充算法主要采用了先用自带的库函数描点，并将描出来的点依次存入栈中保存，进行连线时再出栈进行点与点之间的连接。

对像素点是否在直线所包围区域内的判断：采用y = 常数的扫描线对整个界面扫描，若与图形有偶数个交点，则分别将交点间的点上色。

**实验项目：** Bezier曲线

# 实验目的(Objects)

1. 实现一个绘制3次贝塞尔曲线的程序；
2. 理解贝塞尔曲线的定义方式，会推导任意阶数的贝塞尔曲线方程。

# 实验内容 (Contents)

1. 实现一个绘制分段3次贝塞尔曲线的程序；通过鼠标来交互的给出4个控制点，然后利用这个4个控制点，画出贝塞尔曲线；
2. 每段贝塞尔曲线之间应该是光滑的连接，即满足C0和C1的连续；
3. （附加要求）用户可以通过拖动控制点来修改贝塞尔曲线；可以制造尖点（即C1不连续的点），也可以让某个尖点恢复C1连续。

# 实验内容 (Your steps or codes, Results)

Bezier曲线核心算法：

// 存储x, y结点的坐标数组

private static Point2D[] keyPointP;

// 存储x, y, weight, height的点数组

private static Ellipse2D.Double[] keyPointE;

// 所需要的点 ( >= 3)

private static int keyPointNum;

// t的单次偏移量 t越小 画的曲线越充实

private final static double t = 0.001;

// 定义权重

private final static int WEIGHT = 8, HEIGHT = 8;

// 所显示的蓝色辅助连线

private static boolean flagShow = true;

// 二次贝塞尔曲线 [由 (A B C) 或 (B C D) 三个点构成一条曲线]

// P(1, 1) : (1-t) \* A + t \* B

// P(2, 1) : (1-t) \* B + t \* C

// P(1, 2) : (1-t) \* P(1, 1) + t \* P(2, 1) =

// pow(t, 2) \* C + 2 \* t \* (1 - t) \* B + pow(1 - t, 2) \* A

// P(2, 1) : (1 - t) \* B + t \* C

// P(3, 1) : (1 - t) \* C + t \* D

// P(1, 3) : (1 - t) \* P(2, 1) + t \* P(3, 1) =

// pow(t, 2) \* D + 2 \* t \* (1 - t) \* C + pow(1 - t, 2) \* B

if (keyPointNum == 3) {

double x, y;

g.setColor(Color.RED);

// t 从一个端点自增到另一个端点 (一次自增 0.001)

for (double k = t; k <= 1 + t; k += t) {

double r = 1 - k;

x = Math.pow(r, 2) \* keyPointP[0].getX() + 2 \* k \* r \* keyPointP[1].getX() + Math.pow(k, 2) \* keyPointP[2].getX();

y = Math.pow(r, 2) \* keyPointP[0].getY() + 2 \* k \* r \* keyPointP[1].getY() + Math.pow(k, 2) \* keyPointP[2].getY();

g.drawOval((int) x, (int) y, 1, 1);

// g.drawLine((int) x, (int) y, (int) x, (int) y);

}

}

// 三次贝塞尔曲线 (由两根二次贝塞尔曲线构成) (由 A B C D 四个点构成一个曲线)

// 采用 Cnn高阶导数降幂公式 (莱布尼茨公式)

// P(1, 4) : (1 - t) \* P(1, 2) + t \* P(1, 3) =

// C(0/3) \* pow(1 - t, 3) \* A +

// C(1/3) \* pow(1 - t, 2) \* t \* B +

// C(2/3) \* pow(t, 2) \* (1 - t) \* C +

// C(3/3) \* pow(t, 3) \* D

if (keyPointNum == 4) {

double x, y;

g.setColor(Color.RED);

// 同上 自增

for (double k = t; k <= 1 + t; k += t) {

double r = 1 - k;

x = Math.pow(r, 3) \* keyPointP[0].getX() + 3 \* k \* Math.pow(r, 2) \* keyPointP[1].getX()

+ 3 \* Math.pow(k, 2) \* (1 - k) \* keyPointP[2].getX() + Math.pow(k, 3) \* keyPointP[3].getX();

y = Math.pow(r, 3) \* keyPointP[0].getY() + 3 \* k \* Math.pow(r, 2) \* keyPointP[1].getY()

+ 3 \* Math.pow(k, 2) \* (1 - k) \* keyPointP[2].getY() + Math.pow(k, 3) \* keyPointP[3].getY();

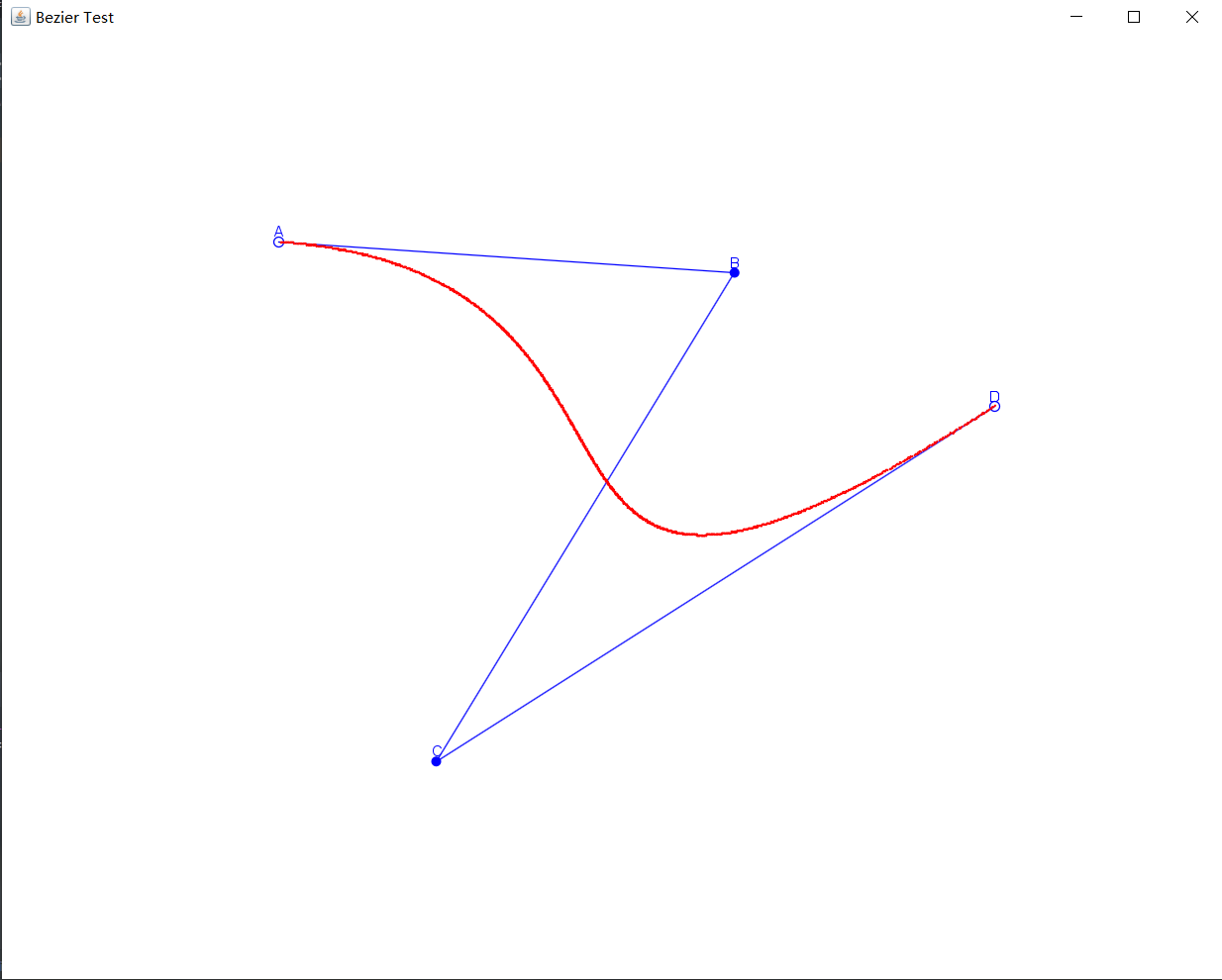
g.drawOval((int) x, (int) y, 1, 1);

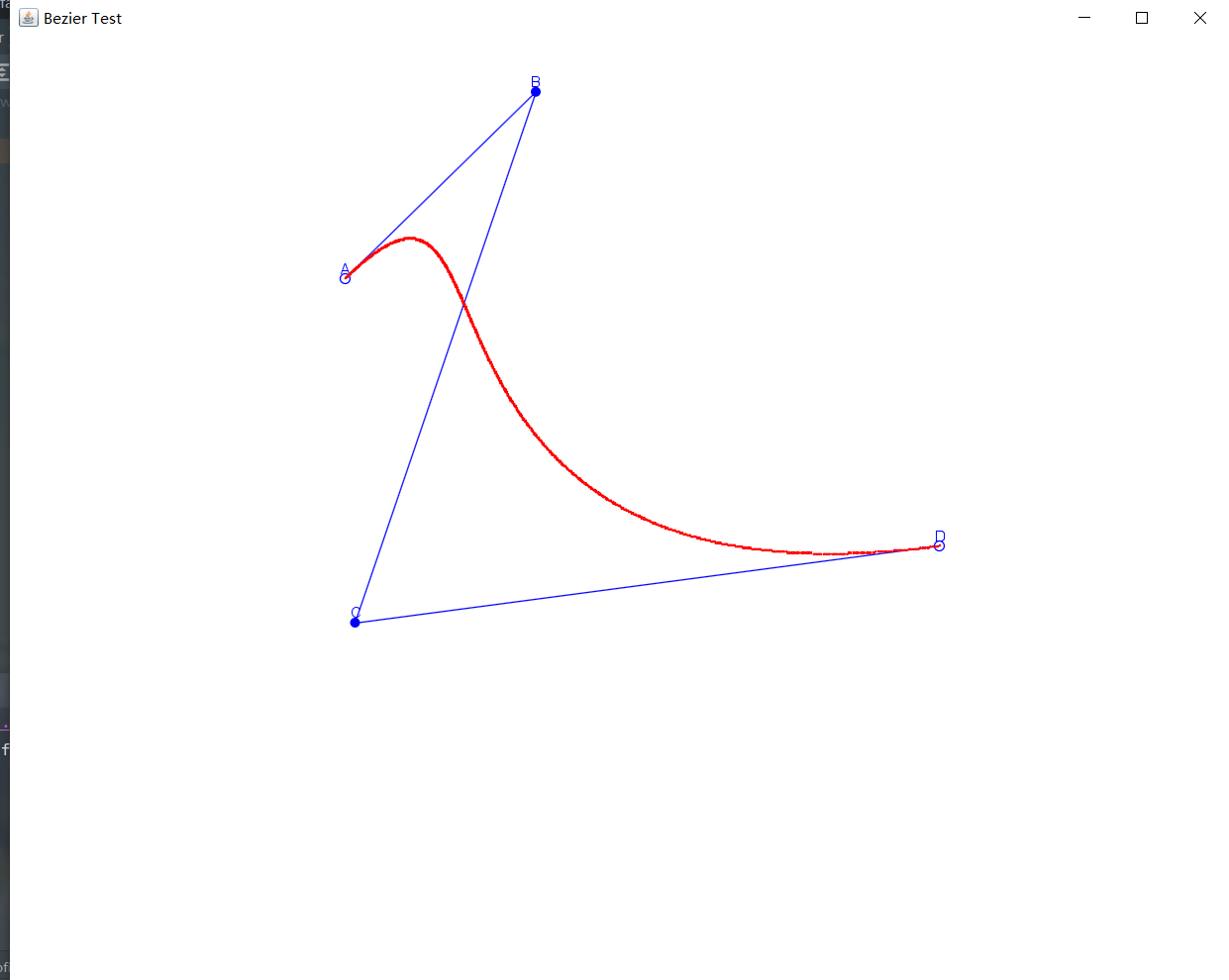
}

}

}

实验截图：





# 实验总结 (Conclusion)

Bezier曲线是由三个点控制，其中有一个为公共点，与其余两点分别组成两条直线，在两条直线中各寻找一点，该点按比例划分直线，连接该两点得到一条新的直线，再按照比例寻找新直线上的一点，该点即为Bezier曲线上的一点。

三次Bezier曲线由四个点控制，其中可以将公共点变为尖点，即可以移动尖点使Beizer曲线可以分段拉伸变换；三次曲线可以理解为由两条二次曲线构成，故在移动一个尖点时，三次Beizer曲线将不会整体受到影响，而是分段移动。

**实验项目：** 图形的几何变换

# 实验目的(Objects)

1. 实现对一个图形进行移动、旋转、缩放的程序。图形可以是点、直线、曲线、多边形或是图像。
2. 掌握使用齐次坐标的意义。

# 实验内容 (Contents)

1. 实现对一个图形进行移动、旋转、缩放的程序。图形可以是点、线、多边形或是图像。要求自己构造矩阵，不能使用库自带的矩阵操作。
2. (附加题)实现一个简单的透视投影变换。
3. (附加题)对图像进行旋转操作。
4. (附加题)实现一种面消隐算法。

# 实验内容 (Your steps or codes, Results)

MVP核心算法：

//translating the local origin to somewhere measured in Window coordinate

void translateInWindow(const QPointF& startPt, const QPointF& endPt){

//qDebug() << centerRef;

double delta[2] = {endPt.x() - startPt.x(), endPt.y() - startPt.y()};

float matrixTemp[3][3];

resetMatrix(matrixTemp);

for(int i = 0; i < 2; ++i){

matrixTemp[i][2] = delta[i];

}

leftMultiply(matrixTemp, transMatLocal2Window, transMatLocal2Window);

}

//translate local origin in Local Coordinate

void setRefCenterLocal(const QPointF& nRefLocal ) {

centerRef = nRefLocal;

}

//rotate each point WRT local origin in Local Coordinate

//this function will change the local coordinate of each point

void rotateLocal(double angle) {

float matrixTemp[3][3];

resetMatrix(matrixTemp);

float t[3][3];

resetMatrix(t);

matrixTemp[0][2] = -centerRef.x();

matrixTemp[1][2] = -centerRef.y();

t[0][0] = qCos(angle);

t[0][1] = -qSin(angle);

t[1][0] = -t[0][1];

t[1][1] = t[0][0];

leftMultiply(t, matrixTemp, matrixTemp);

t[0][2] = centerRef.x();

t[1][2] = centerRef.y();

leftMultiply(t, matrixTemp, matrixTemp);

leftMultiply(matrixTemp, matrixTemp, matrixTemp);

transformLocal(matrixTemp);

}

//translate each point WRT local origin in Local Coordinate

//this function will change the coordinate of each point

void translateLocal(float deltaX = 0., float deltaY = 0.){

float matrixTemp[3][3];

resetMatrix(matrixTemp);

matrixTemp[0][2] = deltaX;

matrixTemp[1][2] = deltaY;

transformLocal(matrixTemp);

}

//rotate the point WRT refCenter measured in Window Coordinate.

//this function will not change the value of the local coordinate of each point

void rotateInWindow(const QPointF& startPt, const QPointF& endPt, QPointF centerInWindow = QPointF(0.f, 0.f)) {

//QPointF centerInWindow = Figure2D::getRefCenterInWindow();

QVector2D vs(startPt.x() - centerInWindow.x(), startPt.y() - centerInWindow.y());

QVector2D ve(endPt.x() - centerInWindow.x(), endPt.y() - centerInWindow.y());

double angle;

vs.normalize();

ve.normalize();

angle = qAcos(QVector2D::dotProduct(ve, QVector2D(1,0))) - qAcos(QVector2D::dotProduct(vs, QVector2D(1,0)));

if(vs.y() < 0 && ve.y() < 0) {

angle = qAcos(QVector2D::dotProduct(vs, QVector2D(1,0))) - qAcos(QVector2D::dotProduct(ve, QVector2D(1,0)));

}

float temp[3][3];

resetMatrix(temp);

//transalte to rotate center : centerInWindow

temp[0][2] = -centerInWindow.x();

temp[1][2] = -centerInWindow.y();

leftMultiply(temp, transMatLocal2Window, transMatLocal2Window);

//rotate

resetMatrix(temp);

temp[0][0] = qCos(angle);

temp[1][1] = temp[0][0];

temp[0][1] = -qSin(angle);

temp[1][0] = -temp[0][1];

leftMultiply(temp, transMatLocal2Window, transMatLocal2Window);

//translate back

resetMatrix(temp);

temp[0][2] = centerInWindow.x();

temp[1][2] = centerInWindow.y();

leftMultiply(temp, transMatLocal2Window, transMatLocal2Window);

}

void scaleInWindow(const QPointF& startPt, const QPointF& endPt, QPointF centerInWindow = QPointF(0.f, 0.f)) {

float dx, dy, sx, sy;

QRectF boundingBox = getBoundingRect();

dx = qAbs(startPt.x() - centerInWindow.x());

dy = qAbs(endPt.x() - centerInWindow.x());

if(dx > dy){ //zoom in

sx = 1.f - 0.1f;

sy = 1.f - 0.1f;

}

else if(dx < dy){

//zoom out

sx = 1.f + 0.1f;

sy = 1.f + 0.1f;

}

else{

sx = 1.f;

sy = 1.f;

}

float temp[3][3];

resetMatrix(temp);

//transalte to rotate center : centerInWindow

temp[0][2] = -centerInWindow.x();

temp[1][2] = -centerInWindow.y();

leftMultiply(temp, transMatLocal2Window, transMatLocal2Window);

//rotate

resetMatrix(temp);

temp[0][0] = sx;

temp[1][1] = sy;

leftMultiply(temp, transMatLocal2Window, transMatLocal2Window);

//translate back

resetMatrix(temp);

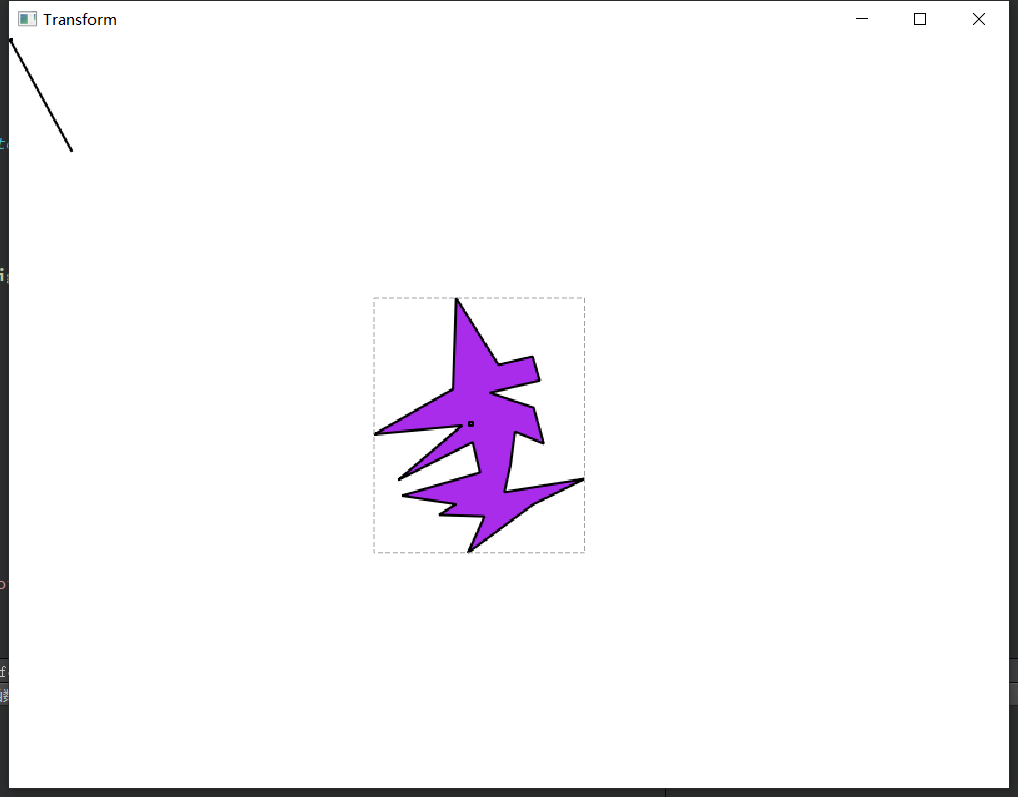
temp[0][2] = centerInWindow.x();

temp[1][2] = centerInWindow.y();

leftMultiply(temp, transMatLocal2Window, transMatLocal2Window);

}

实验截图：



投影透视核心算法：

QPolygonF getXOYOrthoProjection() const{

QPolygonF polygon;

for(const QVector4D & p : ptList){

polygon << QPointF(p.x(), p.y());

}

return polygon;

}

QPolygonF getPerspectiveProjection(const QVector3D &pos) const {

QPolygonF polygon;

for(const QVector4D & p : ptList){

polygon << QPointF((p.z() \* pos.x() - p.x() \* pos.z()) / (p.z() - pos.z()),

(p.z() \* pos.y() - p.y() \* pos.z()) / (p.z() - pos.z()));

}

return polygon;

}

面消隐核心算法：

void ZBuffer(QImage &image, const QVector<Patch3D>& patchList, QVector4D eye){

QVector<QPolygonF> planarList;

planarList.reserve(patchList.size());

QVector<int> patchIdList;

for(int i = 0; i < patchList.size(); ++i){

const Patch3D& patch = patchList.at(i);

#ifdef PERSPECTIVE

patchIdList << i;

planarList << patch.getPerspectiveProjection(QVector3D(eye));

#else

if(!patch.isOrthoXOY()) {

patchIdList << i;

planarList << patch.getXOYOrthoProjection();

}

#endif

}

QVector<float> zbuffer;

zbuffer.resize(800 \* 600);

zbuffer.fill(1000000.);

for(int i = 0; i < planarList.size(); ++i){

const QPolygonF& p = planarList.at(i);

QRectF boundingBox = p.boundingRect();

for(int x = boundingBox.left(); x < boundingBox.right(); ++x) {

if(x <= 0 || x >=800 )

continue;

for(int y = boundingBox.top() ; y < boundingBox.bottom(); ++y){

if(y <= 0 || y >= 600)

continue;

unsigned char \*scanLine = image.scanLine(y);

if(p.containsPoint(QPointF(x,y), Qt::OddEvenFill)) {

double z;

if(eye.w() == 0)

z = patchList.at(patchIdList.at(i)).getZValue(QPointF(x, y));

else

z = patchList.at(patchIdList.at(i)).getZValue(QVector3D(eye), QVector3D(x - eye.x(), y - eye.y(), - eye.z()).normalized());

if(z < zbuffer[y \* 800 + x]) {

zbuffer[y \* 800 + x] = z;

QColor c = patchList.at(patchIdList.at(i)).getColor();

scanLine[x \* 3 + 0] = c.red();

scanLine[x \* 3 + 1] = c.green();

scanLine[x \* 3 + 2] = c.blue();

}

}

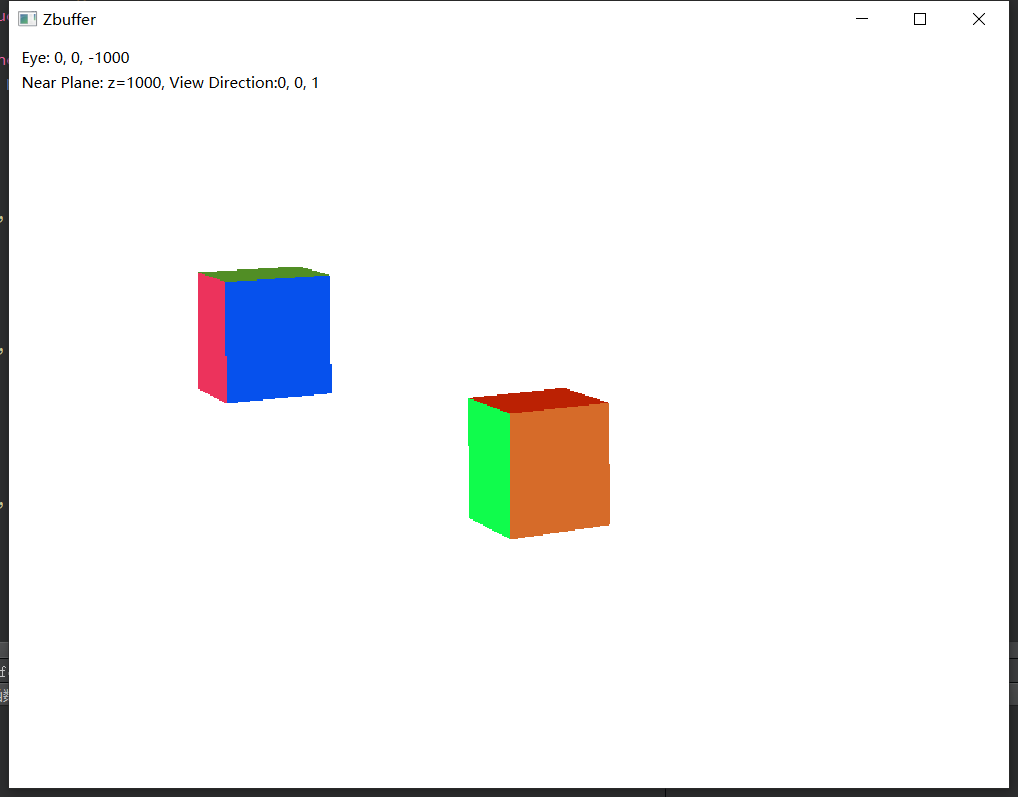
}

}

}

}

实验截图：



# 实验总结 (Conclusion)

如何将一个三维空间的图像投影到二维空间：首先要采用齐次坐标定义一个世界坐标系，将三维空间图形的坐标分别定义为齐次坐标。采用仿射变换Affine中的MVP模式：Model Transform模型变换：将物体通过平移、旋转、缩放移动到需要的三维位置；View Transform视图变换：将相机位置(视线位置)移至需要观察的三维位置；Projection Transform投影变换：分为Orthographic projection 和 Perspective projection，其目的是将先前相机观察到的物体的三维位置投影到二维空间中；先进行透视投影，再进行正交投影。

投影到二维空间后，需要对坐标定义，将图像移至屏幕中央，便需要再运用Viewport Transform视口变换，将图像从原位置移动至直角坐标系的正半轴上；

固定好图像后便可以开始光栅化：shader着色——采用Z-buffer进行深度计算；Bilnn-phong反射模型——计算ADS来实现高光和阴影处理；Texture Mapping纹理映射——将处理好的纹理图对应上图像的u, v坐标进行贴图。

以上便是实现一个三维图形转换到二维图形并进行上色、消隐的过程。